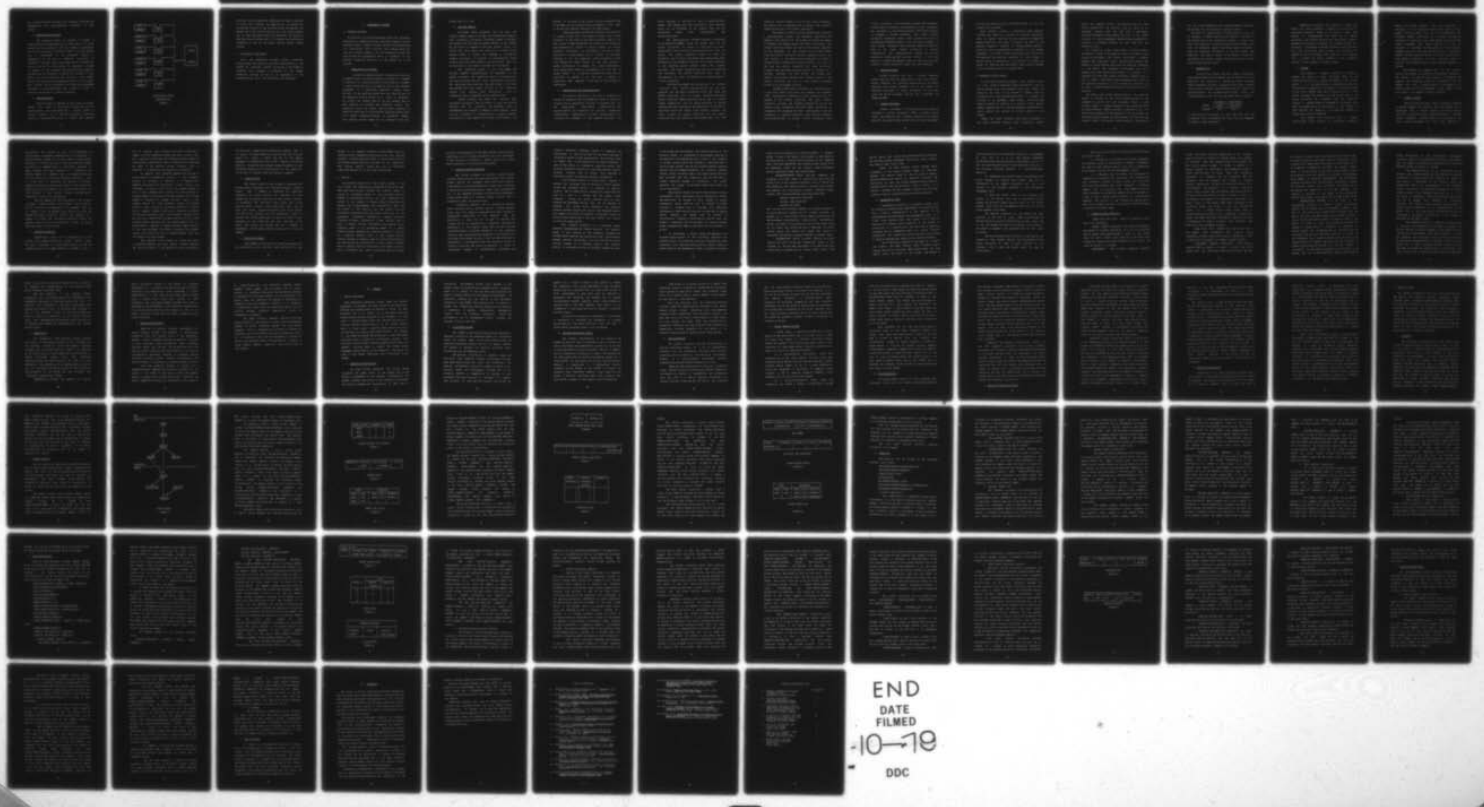
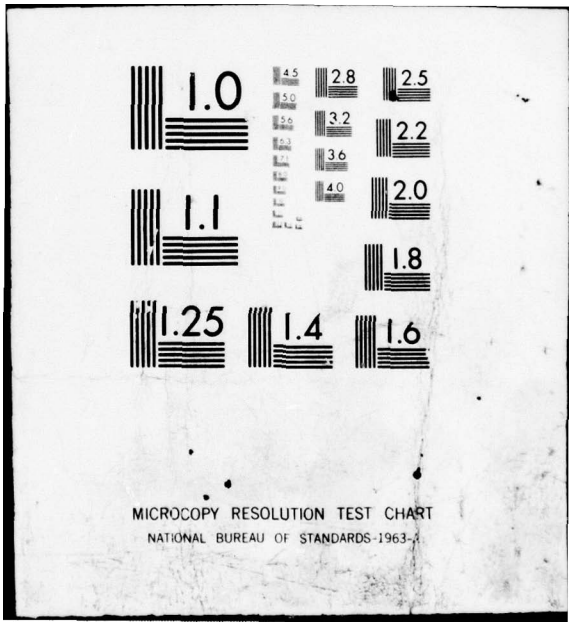
AD-A073 923 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
DISTRIBUTED, SECURE DESIGN FOR A MULTI-MICROPROCESSOR OPERATING--ETC(U)
JUN 79 J S O'CONNELL, L D RICHARDSON

UNCLASSIFIED NL

1 OF 1
AD
A073923



END
DATE FILMED
10-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DISTRIBUTED, SECURE DESIGN FOR A MULTI-MICROPROCESSOR OPERATING SYSTEM

by

James Steven O'Connell

Larry Don Richardson

June 1979

Thesis Advisor: Lt.Col. R.R. Schell

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Distributed, Secure Design for a Multi-Microprocessor Operating System		5. TYPE OF REPORT & PERIOD COVERED (9) Master's Thesis; June 1979
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) James Steven O'Connell Larry Don Richardson		8. CONTRACT OR GRANT NUMBER(s) (12) 90 p.
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE (11) June 1979 13. NUMBER OF PAGES 89
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis applies the state of the art techniques for methodical design of secure operating systems to a distributed, multi-microprocessor environment. Explicit process structure and utilization of virtual environments are the fundamental concepts that form a basis for the design presented. The primary design techniques utilized in the design are segmentation, distributed operating system,

DD FORM 1 JAN 73 1473 (Page 1) EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

251 450

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

security kernel, multiprocessing, "cache" memory strategy and multiprogramming. The resulting design is for a family of distributed operating systems that can provide the power of yesterdays large computer in a microprocessor environment. Security, configuration independence, and a loop free structure are the primary characteristics of the design. The design, although hardware independent, was formulated with the Zilog Z8000 or similar microprocessor in mind.

Approved for public release; distribution unlimited.

DISTRIBUTED, SECURE DESIGN FOR A MULTI-MICROPROCESSOR OPERATING SYSTEM

by

James Steven O'Connell
Captain, United States Marine Corps
B.S., University of Utah, 1971

Larry Don Richardson
Lieutenant, United States Navy
B.S., University of Nebraska, 1973

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1979

Authors James S. O'Connell

Larry D. Richardson

Approved by: Roger R. Schell

Thesis Advisor

Dale L. Cox Jr.

Second Reader

Chairman, Department of Computer Science

D. A. Schrady

Dean of Information and Policy Sciences

ABSTRACT

This thesis applies the state of the art techniques for methodical design of secure operating systems to a distributed, multi-microprocessor environment. Explicit process structure and utilization of virtual environments are the fundamental concepts that form a basis for the design presented. The primary design techniques utilized in the design are segmentation, distributed operating system, security kernel, multiprocessing, "cache" memory strategy and multiprogramming. The resulting design is for a family of distributed operating systems that can provide the power of yesterdays large computer in a microprocessor environment. Security, configuration independence, and a loop free structure are the primary characteristics of the design. The design, although hardware independent, was formulated with the Zilog Z8000 or similar microprocessor in mind.

TABLE OF CONTENTS

I. INTRODUCTION

The microprocessors available today are affordable and powerful computing devices. Applying these resources to various applications, especially those requiring multiple microprocessors, presents a formidable problem. The solution to this problem is a family of operating systems to effectively orchestrate processor and memory management across a wide range of applications. However, such systems have not come from the specialized microprocessor operating systems in use today. Such an operating system family could provide a major reduction of overall system software cost in the microprocessor environment.

In this thesis the substantial body of operating system design principles are applied to a methodical design of an operating system for the microprocessor environment. For realism the Zilog Z8000 microprocessor[1] is considered representative of modern features. Configuration independence, distributed processing, multiple protection domains, multiprocessing and multiprogramming are addressed in the design of a secure operating system suitable for a family of operating systems: ranging from a specialized tactical system to a multi-user time sharing system.

The thesis will also identify meaningful subsets of the design (viz., smaller members of the family) for potential use, and state hardware needed (future development) to

implement the design to its fullest capabilities. The operating system designed in this thesis will be referred to as the SYSTEM throughout the thesis.

A. MOTIVATION

The processing power of microprocessors is increasing. If this power could be effectively coordinated by an operating system it could provide a more affordable and powerful product. In addition, there is a growing emphasis on the protection of information stored and processed in computers; hence, the requirement for a system that also provides information security.

The multi-microprocessor systems in use today suffer performance degradation as more processors (generally a maximum of 4 to 5) are added to the system. Sophisticated crossbar interconnections between processors and memories can reduce this problem. However, there is still a need for a combination of microprocessors and memory that do not suffer massive degradation as more processors are added.

The ability to configure a system to meet a variety of capacity needs is an important feature; however as software becomes an increasing portion of system cost, the ability to reconfigure the system as requirements change without major re-design effort is often an even more valuable feature. For this reason the design technique of resource virtualization will be applied as a way to realize configuration independence.

B. BASIC ELEMENTS OF DESIGN

The SYSTEM is composed of a supervisor and a security kernel[2]. The supervisor supports user services (dynamic linking, discretionary security, demand memory management and a hierarchical file system). The security kernel controls the physical system resources (processors, memory, and external devices) to provide virtual resources for the supervisor.

1. Process Structure

A process within the computer system is an internal representation of the computational task of a user utilizing the system. Each process is characterized by an execution point and an address space. Attributes of each process include a security class authorization and a unique identifier that corresponds to the user. By supporting distinct, explicit processes the operating system allows an application to be divided into several cooperating parts. Such a process structure leads to simpler more effective software.

2. Segmented Virtual Memory

Segmentation involves separating all stored information into discrete packages called segments. Each segment has attributes such as security class and access (read or write) permissions. A process' address space is a collection of segments. Segmentation is used by the

supervisor to present the user a random access virtual memory. Copies of all segments are kept on secondary storage until actually referenced, at which time room is made for it in main memory, possibly by removing another segment from memory. This demand memory management is done within the supervisor. The supervisor views a non-random access virtual memory. By presenting the supervisor and the user with virtual environments the kernel establishes configuration independence for them.

3. Distributed Operating System

The address space of each process has three domains (user, supervisor and kernel). The domains form sub-sets of the address space by limiting the segments that can be accessed when the process' execution point is within a given domain. The operating system is part of each process. It is distributed throughout all the processes in protected domains (supervisor domain and kernel domain). Maximum access is in the kernel domain. It is the most priviledged, and the traditional "privileged instruction' can be executed only in the kernel domain. Only the kernel domain has access to system wide data bases.

The kernel domain creates an extended machine for the supervisor and is supported by system processes. The supervisor is less priviledged but provides the user domain with certain common services such as discretionary security and virtual memory. It should be noted that by distributing

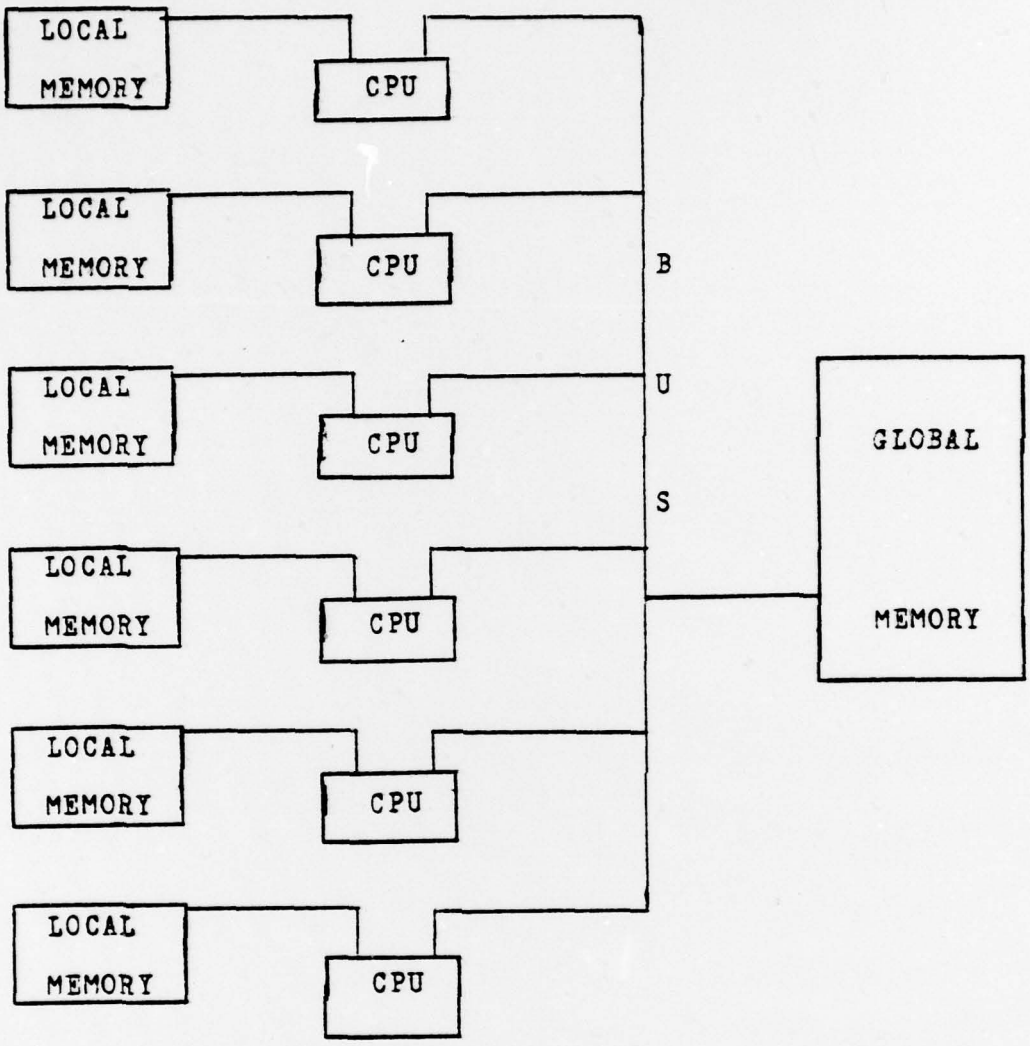
the operating system throughout all processes, services are independently (and simultaneously) available to each process.

4. Processor-Local Memory

The operating system is designed to support a multi-processor configuration with a local memory in close proximity to each processor. The local memory is addressable only by that processor. In addition there is a global memory that is addressable by all processors (Figure 1). Segmentation is the key to effective allocation of information between local and global memory. Problems can arise in the use of a local memory. If a process is allowed to execute on any processor then each time the process is switched from one processor to another the contents of local memory must also be switched. Thus the use of local memory implies that general multiprogramming should not be allowed. This problem can be alleviated by allowing multiprogrammed processes to be semi-dedicated, that is make an effort to restrict the process to a certain processor.

5. Security Kernel

Security cannot in general be built around a present system (i.e., added to) but rather a system must be built around security. Yet today there are a limited number of "secure" systems. One of the main obstacles in providing security is verifying the system is secure. The recently



LOCAL/GLOBAL MEMORY

MULTIPLE CPUs

FIGURE 1

developed security kernel[2] technology has made it possible to solve this problem. By keeping all the things that provide the security in the security kernel and keeping the things that do not involve security out, the security kernel can be kept relatively small and verifiable. The desire to keep the security kernel small (to simplify the verification procedure) is one of the goals driving several design choices.

C. STRUCTURE OF THE THESIS

First, the fundamental concepts (process structure, virtual memory and security) and their relationships to the SYSTEM are discussed. Second, the design of the SYSTEM is presented. This includes a discussion of the design techniques utilized as well as an explanation of the proposed design. Third, the conclusions are presented.

II. FUNDAMENTAL CONCEPTS

A. PROCESS STRUCTURE

By dividing a job into asynchronous parts and executing these parts as seperate entities significant benefits can be realized. Within a single processor system, the partitioning into asynchronous parts provides "only" design simplicity (and thus software economy). In a multi-processor system the partitioning into asynchronous parts is essential if the parallel processing potential of the system is to be realized.

1. Definition of a Process

A process is characterized by an execution point and an address space. Saltzer[3] defines a process as a program in execution on a pseudo-processor. Each process is assigned a unique identifier and is an explicit entity that requires management. In a distributed operating system, those portions of the operating system that are logically part of the sequential flow of control (viz., locus of execution) are within the address space of the user process. This is made possible by dividing the operating system into procedures which are called like any other procedure. It should be noted that in a distributed operating system there is no "master" assigning processes to processors. Rather, each running process "hands off" its processor to the next

process that is to run.

2. Multiple Domains

To protect these procedures from the user, the process' address space is divided into hierarchical domains: user, supervisor, and kernel. The kernel domain is the most privileged. Only the security kernel executes in this domain and can access all segments within the address space. All system wide data bases are restricted to access by the security kernel to prevent any exchange of information between processes, in violation of confinement[4]. There could be more than three domains, and all domains need not be hierarchical, but three is minimum for this design.

The supervisor domain is less priviledged and excludes segments representating the management of the shared resourses. The supervisor domain is separated from the user to protect the user from inadverently destroying the operating system services. The user domain is the least priviledged. The data bases utilized by the supervisor contain only "process local" information - that is, information that is required by this process alone.

Proper controls and checks are utilized when switching the domains (flow of control) so that the security policies are not violated. The hierarchy could be implemented with rings[5] in hardware. Since hardware rings are not available in microprocessors, separate segment descriptors for each domain can be used, with software ring

changes as was done in the original Multics design[6]. The Zilog Z8000 can use multiple memory management units (MMU) to provide the separate descriptor for each domain.

Operating system procedures generally are permitted to reside within the local memory (possibly ROM) of each processor. In the cases of the security kernel, some of the data bases of these procedures are shared by all processors and therefore will reside in global memory. To prevent undesired intervention by simultaneous accesses to these data bases a locking scheme must, of course, be provided. Choosing to put the operating system procedures in each local memory will "waste" memory but may well provide a higher performance by keeping most memory references to local memory where there is no contention for the BUS to global memory. In a specific instance the choice will be determined by whether or not the cost of memory is significant when compared to the value of the increase in performance.

3. Communication and Synchronization

For parallel processing, a job that is composed of a mixture of sequential and non-sequential tasks is explicitly divided into an appropriate structure of processes that can run concurrently. Inter-process communication and synchronization are necessary for parallel processing. Inter-process communication provides synchronization to coordinate the exchange of data between processes. The

actual exchange is realized by use of a shared writable segment. This segment acts like a mailbox in that messages (data) can be delivered by any process that has the appropriate access (both discretionary and non-discretionary).

The synchronization between processes is supported by the BLOCK and WAKEUP, which are kernel calls to the traffic controller. It should be noted that the P and V semaphores[7] are useable for synchronization but were not chosen. The traffic controller concept is taken from Saltzer[3], and his block and wakeup have demonstrated their usefulness in his design for Multics. The traffic controller is the operating system (kernel) module that manages processes. The traffic controller has the job of scheduling user processes. The traffic controller does this by multiplexing the users processes onto a limited number of virtual processors.

The BLOCK and WAKEUP are primitives of the traffic controller that provide synchronization for the user processes. How the user's procedures invoke the BLOCK and WAKEUP primitives will, of course, determine the actual process structure. These primitives can be used to provide simple cooperation, such as mutual exclusion, or complex interactions when required by the application. A process can only block itself and cannot block another process. The block invokes the traffic controller and the traffic controller puts that process in the blocked state and then

schedules another process to run on that virtual processor. The process that is scheduled next is based on the specific scheduling policy of the traffic controller.

The wakeup is used to provide asynchronous processes a synchronization signal. The parameter passed with the wakeup is the process ID of the process for which the wakeup is intended. The wakeup invokes the traffic controller. The traffic controller checks the state of the process specified by the parameter. If that process is not in the blocked state the traffic controller returns, otherwise he will put that process in the ready state and determine if there is another process running with a lower priority. If this is the case the traffic controller will send the virtual processor that the lower priority process is running on a pre-empt interupt, and then return. The process that receives the pre-empt interupt will transfer control to the traffic controller who will in turn schedule the ready process with the highest priority.

Another SYSTEM module concerned with synchronization is the inner traffic controller. This manages the hardware (real) processors to create the virtual processors that are managed by the traffic controller. The inner traffic controller provides the interface between the virtual and physical (real) processors. The inner traffic controller is responsible for assigning the small, fixed number of virtual processors to physical processors. Each physical processor has associated with it several virtual processors. Some

virtual processors are multiplexed between users processes by the traffic controller. The remaining virtual processors are allocated to the system processes. Each system process is assigned a virtual processor. The inner traffic controller determines which virtual processor will run on the physical processor based on the priority assigned to each virtual processor. The primitives SIGNAL and WAIT are used by the inner traffic controller to provide communication and synchronization between the virtual processors. SIGNAL and WAIT are very similar in form and function to BLOCK and WAKEUP, except for the fact that they relate to virtual processors rather than user processes.

4. System Processes

System processes are used to perform operating system functions that are asynchronous to each user process. System processes are typically responsible for the shared resources. The system processes are in the kernel and therefore permitted to access information of any access class. The system processes include the I/O_MANAGER and MEMORY_MANAGER.

5. Process Switching

Process switching is the removing and assigning of processes to virtual processors. When a process switch occurs the execution point (internal registers) and address space of the process being removed must be saved (unloaded),

and then the execution point and address space of the new process must be loaded.

Some systems utilize a descriptor base register (DBR)[6,p.12], which is a pointer to multiple descriptor lists in memory - one list for each process. To change the address space you only need to switch the DBR in the physical processor. However, in microprocessor systems a descriptor list is implemented as registers in the memory management unit (MMU). Process switching can be costly when MMU registers are saved and restored for each change in address space. Alternatively, it is possible to increase the number of MMUs and then the address space could be changed by just switching control to another MMU.

B. SEGMENTED VIRTUAL MEMORY

In many memory handling schemes a user process cannot run until there is sufficient memory available to load its entire address space. This requires large main memory and restricts the size of the process's address space. An alternative is to use the operating system to produce the illusion of an extremely large memory. Since the large memory is merely an illusion, it is called virtual memory. Demand segmentation is a memory management scheme which is used to realize the concept of virtual memory in this design.

Memory has three different views which corresponds to the three different domains (user, supervisor, kernel)

within the computer system. Starting with the user, each view is derived from the previous view by means of an "extended machine" view. The user sees a practically unlimited segmented virtual memory. The user is no longer involved in memory management. Demand memory management is utilized to interface between the user view and the supervisor view.

The supervisor views a fixed amount of virtual memory. The memory is fixed by the physical memory allocated to each process by the kernel. The kernel establishes a mapping between the supervisor memory and the kernel memory. The memory is virtual because there are only absolute addresses in the kernel. The supervisor multiplexes the user's segments onto this fixed virtual memory in response to a hardware fault when the process references a segment that is not in memory. The demand memory management was placed in the supervisor because it is not involved with security and we want to keep the security kernel as simple and small as possible.

The kernel views a fixed physical memory. The physical memory is limited by the local memory available to the processor for use by the user processes. There is some minimum amount of memory required by the operating system for each processor. Before a process is elgible to run, its fixed virtual memory (of the supervisor) must be mapped into the fixed physical memory of the processor. We then call the process "loaded". The kernel's memory manager is responsible

for the proper mapping as the processes address spaces are multiplexed onto the processor's physical memory.

The idea is to require that a limited amount of the job be resident in memory. When the user requests a portion of the process that is not currently in memory, a fault will occur. The supervisor, using the demand memory manager, must find the requested segment and decide where it wishes to place the requested information in virtual memory. The supervisor then sends a request to the kernel to bring this information into memory, thereby repairing the fault so that normal processing can resume.

1. Segmentation

In most micro systems, the user cannot effectively share memory because the different uses of memory can not be specified. The inability to specify the memory use makes memory management difficult, especially when there is memory local to each processor. The different uses are denoted by shared/unshared and writeable/non-writeable (read). The following matrix lists the uses and where they may reside.

	writeable	non-writeable
shared	global	local/global
unshared	local	local

If the memory can be divided by uses and each part has attributes which distinguish the uses, then the management of memory is made reasonable.

Segmentation provides the ability to divide the memory into parts (segments). A segment is a collection of information important enough to be given a name. Each segment is distinguished from others by its logical attributes, that provide the basis for the desired control. Segmentation provides a mechanism for a limited portion of a processes' information to reside in memory at any one time. This also facilitates easy movement of information by segment in and out of memory. The collection of all segments that a process may access (whether or not in physical memory) is what composes its address space.

2. Loading

The loading of a segment consists of finding a segment and making it known (discussed later) to the requesting process (viz., adding the segment to the address space). It is the added feature of segmentation that this loading may be delayed until the segment is actually needed. At that time a segment name can be transformed into a file system pathname. The pathname can then be resolved into the unique identifier for a segment. Then the supervisor requests a segment number be assigned by the kernel, which makes the segment known to the process. If the segment is then required for execution it is physically loaded into memory when actually referenced.

Each segment has associated with it a segment descriptor[6] which contains its attributes (address in

memory, size, access allowed). Since this descriptor is referenced by the hardware at each access request to this segment, then the memory uses can be distinguished. The different segment descriptors of a process can then be contained in a descriptor list. This design utilizes the MMU (memory management unit) which consists of a set of registers to implement the descriptor list. Each register (segment descriptor) contains the descriptor of a particular segment. The MMU registers retain the distinct attributes of each segment at execution time and, therefore, makes it possible for another process to share selected segments, if desired.

The dynamics of a segment fall into two classes, physical and logical. An example of the physical dynamics is the request of a user for write access to a currently used segment. The operating system can physically move the segment from local to global memory so the segment can be shared without the user's knowledge. A stack segment whose size varies is an example of logical dynamics.

3. Dynamic Linking

When a procedure segment makes an external reference to another segment, the address of the later segment must be determined. This is called linking, the constructing of executable instructions that achieve references to externel objects (segments). Linking need not be completed at load time. It can be postponed until the actual reference is

encountered. This waiting to link, until referenced, is called dynamic linking[8]. Segmentation is not necessary to achieve dynamic linking, but it helps. When a process begins execution, it should not have to find and bring into memory any more of its segments than is absolutely necessary to begin running. The mere presence of a reference to an external segment in a segments text is no guarantee that the flow of control will touch this reference. Therefore, there is little point to undertake the expense of finding a segment and making it known unless there is some significant expectation that that segment will be referenced during the time allotted to that process. Dynamic linking permits unnecessary linking to be eliminated.

Once the segment has been made known to the process (assigned a segment number), even though it may be moved in and out of memory, the references to this segment need not be changed since the segment number remains the same. The segment descriptor is used to reflect the presence of a segment in memory and the current address in memory. The segment looses none of its attributes by virtue of having been made known to this process.

4. Information Sharing

Segmentation allows direct addressability by the process to any segment within the process' address space. The basic advantage of direct addressability is that the copying of data is no longer mandatory. A segment is also a

unit of sharing. This eliminates the need to duplicate a segment for each requesting process and saves memory. Even more important is the idea that sharing provides a means of inter-process communication. This is important for realizing the power of the explicit process structure, that is essential to an effective multi-processor environment.

In general each procedure segment must be pure to ensure sharing is implemented correctly. A pure procedure operates on variables in registers or in separate data segments associated with the process. It never stores data internally, nor does it alter itself. The linkage segment is such a data segment used to support the pure procedure. A linkage segment is associated with each process. The linkage segment is composed of linkage sections. There is one linkage section for each procedure segment. The linkage section is used to place all alterable information (linkage faults, segment numbers, other static temporary variables) for the pure procedures. Thus, the processes' segments which are pure may be shared while linkage sections must be unique to each process. The fact that the linkage segments are not shared makes it possible to assign different segment numbers to the same procedure in different processes since segment numbers occur explicitly only in linkage segments, that may be different for each process.

The approach in this design is to place the copies of requested segments into local memory, thereby reducing the data bus traffic. If the read-write access requirements

are such that a segment must be physically shared, then it is placed in global memory and every process that is given access will access it there. The key to this memory management is segmentation that keeps a segment's attributes explicit. The kernel can properly manage placement in local and global memory with no intervention from the supervisor or the user to "declare" that the sharing is needed.

5. Access Control

The access control in this design is separated into discretionary (supervisor) and non-discretionary (security kernel). When a segment is requested the supervisor references the access control list attribute for that segment and the access authorized for that process (subject) is determined. The supervisor then passes this to the security kernel so that a non-discretionary check can be made. The kernel compares the access class of the segment with that of the process and the appropriate access is allowed. This access authorized is always the lesser of that requested by the supervisor and that permitted by the kernel. The access one process has for a segment is independent of the access another process has for that same segment.

6. Functional Subsets

Some members of the family of operating systems will not include all of the functions made available by this

design. As an example, consider a family member (e.g. for tactical system) supporting applications that are entirely resident in memory and pre-linked. It would require none of the virtual memory functions provided by the supervisor. This design readily allows this sort of functional subsetting because of its loop free structure[9].

C. SECURITY

The increased capability of the computer system in the last decade has dramatically increased its possible uses. Many users have actively allowed the computer system to assume an increasing number of jobs upon which the user depends to successfully function. As more dependence was placed on the computer it became evident (regrettably by example) that a knowledgeable user (employee of a user) who has access to the computer also has access to all the information contained within the system. Users such as the government (classified information), banking facilities (transfer of funds), corporations (trade secrets) have a need to protect certain information from specific users; therefore, there is an increasing demand for a secure computer system. Designating a specific computer to only run at a specific security class or only running certain security classes at specific times has proven unsatisfactory for the user who has information at many access classes. What is commonly called a "multilevel" environment is one in which information and users at different security classes

can exist simultaneously on the same computer system without permitting a user to access information he is not authorized to use. One goal is to design a system which will allow secure operation in a multilevel environment.

1. Computer Security Problems

The initial attempts to provide a secure system involved adding security onto existing systems. This proved largely useless for designers were intuitively trying to block methods of would-be-penetrators rather than providing a technically sound system design. These futile attempts[10] led to the emerging technique of methodically designing a secure system based on a security kernel derived from a mathematical model (discussed later).

Information security can be provided by external and/or internal control. External control includes guards, watch dogs, door ciphers or anything which would prevent an unauthorized penetration of the compound. Once the penetration is made, the pot of gold is exposed. The internal control is concerned with preventing unauthorized penetration of the computer system. This involves insuring the effectiveness of internal mechanisms in the operating system so that only authorized exchanges of information in a multilevel environment can occur. This includes providing no information to unauthorized users and consistent replies to security violations. The latter is necessary to insure no inadvertant leakage of information[4] concerning the

internal mechanisms. External control is expensive and human-prone. It does not provide for the secure sharing of information needed by many applications, thus forcing users to forego many of the capabilities of modern computers. A goal of this thesis is to design an operating system that provides information security by utilizing internal control. External controls are, of course, still required to physically protect the computer system's information.

The reference monitor is an abstraction created to present the conceptual idea of providing a secure computer system. The reference monitor is composed of subjects, objects, and an access matrix. Subjects are system entities such as a user or a process that can access system resources. Objects are system entities such as data, programs and peripheral devices that can be accessed by subjects. The access matrix represents the permitted accesses between subjects and objects. The reference monitor must support the ability of subjects to reference objects as per the access matrix and it must also support the ability to alter the access matrix.

The security kernel[2] is a relatively recent technical breakthrough for computer security. The security kernel is that portion of the computer's hardware and software which enforces the authorized access relationships between subjects and objects. It is the realization of the abstract concept of a reference monitor. The software portion of the kernel acts as an interface between the rest

of the system and the hardware. The software content of the security kernel is influenced by the hardware features of the processor. The underlying idea is that if the hardware is proven correct and if the software is kept small and it can be proven correct, then we can provide internal security controls that are effective against all possible internal attacks. Global variables such as the unique identifier have been excluded from the supervisor. This has been done to prevent undesired leakage of information. The global variables are placed in the kernel where their proper use can be verified[11].

The security kernel must meet three essential design requirements. First, the kernel must be tamperproof. Second, the kernel must be invoked on every attempt to access information. Every reference must be checked by either software or hardware that is provided with sufficient information to make correct decisions on granting or denying access. Finally, the kernel must be subject to certification. "Subject to certification" implies that the kernel's correctness must be proveable in a rigorous manner using a mathematical model as the basis for the criteria to be met.

In developing a secure system the approach to be followed should consist of the following: determine the security policy to be enforced, develop a mathematical model consistent with desired security policy, design a security kernel based on the mathematical model, implement the design

using available hardware and required software. A computer system is said to be "secure" with respect to some specific security policy. A security policy consists of the external laws, rules and regulations that establish what access is to be permitted. There are two distinct types of security policy: non-discretionary and discretionary.

NON-DISCRETIONARY POLICY involves checking the requested (viz.,the object's) access class (oac) with the access class of the (subject) requestor (sac) to insure they are compatible. Each system contains a lattice structure[12] that defines the relationships between different access classes. The following defines the access permitted:

sac=oac, read/write permitted

sac>oac, read permitted

sac<oac, no access

The lattice can be totally ordered (all classes related) or it can be partially ordered (not all classes related). An example of a policy with totally ordered classes would be the government classification (unclassified, confidential, secret, top secret) of information, oac and the access class of its' users, sac, called the user's clearance. For such a lattice policy the system must insure that access to classified information is always confined to cleared users.

DISCRETIONARY POLICY involves checking an access control list (ACL). If the user requesting access is not included on the ACL then the access is not permitted. This allows users to specify who can access their files. This

policy really lies within the non-discretionary structure and provides further refinement. This policy would reflect the "need to know' rule of DOD.

There are many distinct system designs which correspond to the almost endless number of policies; however, the current state of the art allows a simple, uniform mechanism for nearly all practical policies. The implication is that the kernel designer does not have to concern himself with the particular security policy of a specific customer. He must, however, consider the two broad classes of policy: discretionary and non-discretionary.

2. Mathematical Model

A mathematical model[13] is a powerful design tool for formally translating the requirements of security policy into a precise representation of the behavior of the corresponding security kernel. The mathematical model is a finite state machine model that gives a set of rules of operation for making a state transition. If the system is initialized to a secure state, then the rules of operation guarantee that all subsequent states are secure. Previous research[14] has proven that security kernels whose design is based on mathematical models can be certified correct.

Two of the basic elements of the model are subjects and objects. The model defines types of accesses that a subject may have to an object. These access types are read and/or write. The state of the system with respect to

non-discretionary and discretionary security is represented by four sets (b, m, f, h). This design implements non-discretionary security policy in the kernel (sets b, f) and the discretionary policy in the supervisor (sets m, h). The folowing discussion pertains to non-discretionary security.

b - represents the current access relationships that exists between all subjects and objects. This set is represented by the segment descriptor list, viz., the contents of the hardware registers in the MMU (memory management unit).

f - gives the access class of all subjects and objects in the system. This set is distributed in this design: the process's access class is found in the active process table (APT) and the segments access class is in the active segment table (AST).

The desired properties of the system are then realized in the form of rules. These rules enforce the desired security policy by manipulating the sets which may or may not change the state of the system. If the state of the system is changed it must guarantee that the new state is secure.

The discretionary security policy is enforced in the supervisor. This design decision was made because of the lesser importance of "need to know" controls to the military, and to keep the kernel small for ease of verification.

The sets which are used to enforce the discretionary policy are m and h.

m - corresponds to an access matrix which represents the potential access of the subjects to objects (implements the "need to know" security policy). This set is represented by the access control list for the segment (object).

h - indicates how the objects are hierarchically organized in a directory tree structure. The hierarchical tree structure consists of nodes, leaves, and a root from which the tree eminates. The nodes represent a directory segment (list of attributes for other segments) and the leaves represent non-directory segments (data or procedure). A user is free to create either directory or non-directory segments. The ability to add directories implies that a user, if he chooses, can add to the overall system hierarchy a subtree of arbitrary depth.

3. Properties And Conditions

There are a few basic security properties which need to be considered:

SIMPLE SECURITY CONDITION- this condition addresses the problem of security compromise. If in set b all subjects have an access class greater than or equal to the access class of their objects, this condition is satisfied. This insures the subject only reads information at or below the class for which it is cleared.

CONFINEMENT - this property addresses potential

(rather than actual) security compromises. If all subjects could be trusted to perform in a proper manner (with respect to security), then this property would not be needed. The fact is that unless a program is proven to behave in a certain fashion as described by the mathematical model or formal specification, we cannot make any statements concerning its behavior. We must therefore make the assumption that the programs will attempt to violate security regulations. Subjects are therefore assumed to be untrustworthy. The potential for a security compromise occurs when a subject has simultaneous read access which is at class a and write access at class b (class a >class b). For example, the potential for compromise is realized if two events occur: (1) the subject reads secret information from the secret object and writes it into the unclassified object. (2) a second subject whose access class is unclassified gains access to this (nominally unclassified) object and reads the secret information. There are two ways of preventing this type of situation from occurring: high water mark and confinement property.

High Water Mark - upgrade the class of the file to the highest class requested. This solution, while technically correct, would over classify information so that it would not be available to normally cleared subjects.

Confinement Property (*-Property) - this property requires that all objects to which a subject has write access have the same access class as the subject and that

all objects to which it has read access have an access class less than or equal to the access class of the subject. Since a subject will always have write access to some object if it is to perform a computation, we define the current access class to be that class at which the subject wishes to have write access. Since all subjects are assumed untrustworthy with respect to security requirements, the confinement property eliminates the certification requirement outside the security kernel. This eliminates the immense job of certifying the supervisor and the user programs. This property is enforced in the kernel by not allowing any subject write access to an object with a lower access class.

COMPATIBILITY PROPERTY - If an object in the hierarchical structure is inferior (child) to an object (parent) and the access class of the parent is greater than that of the child, then a subject with an access class the same as the child can never access that information since it can not access the access control list which is kept in the parent. In order to avoid this problem we introduce the concept of "compatibility". A hierarchy is compatible if access classes are non-decreasing as one moves down the hierarchy from the root. The access class of an object in the hierarchy must always be greater than or equal to the access class of its parent. Since the root has no parent its security attributes are implied (viz., are the "lowest" of any object). In this design compatibility is enforced in the kernel, but not in the traditional sense of enforcing the

access relationship of the parent/child hierarchical structure. There is no hierarchical structure in the kernel. When the segment is created the compatibility is implicitly enforced before the request is allowed.

The reference monitor is an abstraction of the hardware and software mechanisms that mediate all attempts by subjects to access objects. The decision to permit or deny access is determined by the security kernel. The mathematical model is an interpretation of the reference monitor abstraction and describes the behavior of a secure system in terms of four component data bases (b, m, f, h) and rules of operation. These rules specify how the data base may be changed, they represent an "authorize" operation. The security kernel can only allow subjects to access objects as permitted by its representation of the model's set b. The data base of the security kernel must correspond to the model's data base and can only change as permitted by the model's rules.

The reference monitor of a physical computer system is realized by a combination of software and hardware. The portion required in software depends on the capabilities and limitations of the hardware. There may be objects to which the hardware can not properly control access and there may be alternative representations of the same security state. Either one of these situations require a kernel function that does not change the security state. In the former case there would be one or more functions to permit interpretive

access to an object; in the latter there would be functions for changing the representations of the security state without changing the actual state.

Thus the functions of the security kernel software[2] fall into three classes that correspond to the fundamental operations of authorize, access, and null: (1) functions that correspond to the rules of the model, thus changing the security state; (2) functions that implement a part of the reference monitor by allowing interpretive access to objects as permitted by the current security state, thus complementing the hardware access controls and (3) functions that change the representation of the current security state.

4. Segmentation

The mathematical model addresses abstract subjects and objects. In this design subjects are the processes and the principal information objects are segments. Processes (subjects) can only access segments (objects) as permitted by the access controls. Every segment has associated with it logical attributes (access class, size, read/write permission) which are made visible at the time of actual reference to the information. By including access control as part of the logical attributes, a way to control access to the information in the system has been provided. Only 'authorized' accesses are allowed.

Segmentation provides the mechanism so that all

online information stored in the system is directly addressable by a processor and hence available for direct reference by any computation. A basic advantage of direct addressability is that users can physically share a single copy. A concern which arises from sharing is that information may be passed illegally between users. This is prevented by the enforcement of the confinement property and the simple security condition. The copying of data is no longer mandatory as many users can share a single copy with controlled access.

5. Hardware Requirements

There are no absolute hardware requirements for secure computer systems, any hardware is theoretically acceptible. Given the current state of the technology, however, certain hardware features are essential if we are to build efficient secure systems[2]. These essential features reduce and simplify the software portion of the security kernel. Reduction and simplification of software at the expense of additional hardware is necessary because producing proveably correct software and hardware in the security kernel is a necessity to achieve computer security.

One of the essential features is support for a segmented memory. Segmentation allows all information in the system to be stored in one type of object, the segment. Having to support only a single object type simplifies the kernel. Segmentation allows all information in the system to

be compartmentallized into individual packages called segments. Every segment has associated with it access controls as previously mentioned. Only authorized accesses as delineated in the access control list and allowed by the access class are permitted. The address of information is composed of two parts (segment #, offset). It is necessary to efficiently resolve the two dimensional address into an absolute address, therefore segmentation should be implemented in hardware.

The other essential hardware feature is multiple execution domains. This feature is used in most contemporary systems to protect operating systems from applications programs. Strictly speaking only two execution domains are necessary (one for the kernel and one for everything else), but in practice it will still be desireable to continue to protect the operating system from applications software so three domains (kernel, supervisor, user) will be used in this design.

III. DESIGN

A. DESIGN TECHNIQUES

When designing an operating system there are several approaches to consider: top down, bottom up and middle out. Although most designs begin as top down or bottom up they generally end up as middle out. In the design there are several design choices available to the designer. In some cases a certain design choice will preclude the ability to utilize a specific design later on in the system design, while in other cases a specific design choice could be a driving force to dictate other design choices. For example in the SYSTEM the design choice was made to keep the kernel relatively small to reduce the verification process. This particular choice became a heavily weighted factor when, for example, deciding where to support the demand memory management which ended up in the supervisor. Following are some of the design techniques that contributed to the SYSTEM.

1. Resource Virtualization

By using virtual processors and virtual memory throughout the upper levels of the design, most of the design is independent of the physical configuration. The SYSTEM provides the virtual to real binding in the kernel. This permits changing the configuration to meet user or

maintenance requirements without major changes to the system. Since the processes are assigned virtual processors there is no effect on the user when real processors are added or deleted (except for the change in performance). Of particular interest was the ability to add and delete processors to the SYSTEM. More important was to develop a design that allowed good capacity growth with the addition of processors. In general, configuration independence implies that the hardware (processors, memory and peripherals) can be reconfigured without causing any problems visible to the user.

2. Distributed System

The SYSTEM is distributed logically and physically. Logically, portions of the operating system are distributed within the address space of the users process within the supervisor and kernel domains. The use of domains permits the process to maintain its security attributes while interacting with the operating system.

The physical distribution of segments among the individual local memories provides performance (provides high speed memory access and limits BUS contention). The physical distribution allows the tradeoff of memory (viz., multiple copies) for performance. Although one of the potential benefits of segmentation is sharing of pure procedures the choice was made to disregard this benefit when possible (no user has write access). This allows the

segment (viz., a copy) to reside in local memory to reduce BUS contention. The initial hypothesis is that the memory wasted (much of it possibly ROM) is a small price to pay to allow performance to grow well with the addition of processors. This addresses the problem that in typical multiprocessor systems capacity scales poorly because of increase load on the BUS. However, this choice is not fundamental to the design and could be changed to eliminate multiple copies.

Similarly for processors, processing is distributed to processors to eliminate the dependency on a single controlling unit. The system wide data bases are kept in global memory providing access to all processors.

3. Multiple Protection Domains

The foremost consideration in the design of the SYSTEM was security. This is acheived by use of the security kernel technology, and segmentation provides one of the keys to providing security within the system. The set of segments that are accessible is defined as a domain. The conventional two state system does not provide the desired support for a secure system. For this reason the 2-state (and associated 2 domains) is generalized to a hierarchical n-domain system[6]. In the design of the SYSTEM (a minimum of) 3-domains were considered adequate - user, supervisor and kernel. In addition, the design permits that, based on user application, a number of user domains could be supported.

Each domain is in concept similar to a ring[6]. The authorized access of a process is determined by the current ring of execution. The access within the different rings form a set of nested domains. Ring Ø (kernel) is the largest set and ring n-1 is the smallest.

The ring structure with the associated controls provides a means for regulating the information that passes between domains (rings). Cross-ring calls and parameter passing are well defined[15]. When the proper controls are used they allow outer rings to make requests to inner rings, but also protect the inner rings from unintentional or intentional tampering. The ring structure when combined with segmentation provides mechanism for the design of an effective secure system by protecting the secure kernel.

4. Multiprocessing

The process structure provides the essentials for parallel processing: support for a set of assynchronous processes that can communicate with each other. Parallel processing does not require a multi-processor environment. However, in a multi-processor environment parallel processing can provide faster completion of a job.

There are many applications for parallel processing within tactical as well as non-tactical systems. Whenever a job depends on a mixture of asynchronous and synchronous tasks and time is a factor, parallel processing is a possible solution to getting the job done in the allocated

time. By using several processors working on the same job, each doing seperate tasks, the overall time required to do the job can be reduced (provided the job has been structured into explicit processes). In microprocessors where processors are relatively inexpensive and slow, parallel processing may be the answer to keeping the cost down while still being able to complete the job in the required time. The above discussion provides some of the major reasons why the SYSTEM was designed to support parallel processing on multiple processors.

5. 'Cache' Memory Strategy

A cache memory is generally thought of as a small amount of high speed memory that is utilized with a large low speed main memory in a system to construct a memory system that appears to be a larger high speed memory. This appearance of a high speed memory is generally possible as a result of locality of reference[16,p.301].

In a multiprocessor environment, where each processor has its own cache memory, problems arise when accessing shared memory. The main problem being that shared, writable memory cannot be put in a cache. Segmentation allows the assignment of attributes to segments, which provides a way to identify cacheable segments (those segments that are not writable and shared).

In a multi-microprocessor system where BUS contention can become a problem a cache memory strategy

could be quite effective in reducing the number of requests to the main memory, even though the cache and shared memory are the same speed. The main advantage is avoiding access to the system BUS rather than the increase in speed of the actual memory access. The SYSTEM uses the strategy of a cache in the form of a local memory per processor. Now rather than being a copy of what is in global memory the local memory (cache) becomes the place where the data is stored instead of global memory (note that with a cache, global memory need not contain a copy while the information is in the cache).

Each processor has its own local memory which is relatively large in size where cacheable segments are stored. This means that large blocks of data will be moved when a process is removed from one processor and (subsequently) loaded on another processor. In addition a global memory is utilized for shared writable segments (unencacheable segments). Segmentation allows the SYSTEM to utilize the concept of caches and main memory but in the form of local and global memory. The overall reason is the same (speed up memory access), but in the SYSTEM this is achieved by reducing the BUS contention through directing most access to local memory.

6. Multiprogramming

In a system where there are more processes than processors there must be a means of switching processors

from process to process. Some reasons for switching process are: current process completes, a higher priority process is ready, current process is blocked, or current process is waiting I/O. Whatever the reason for switching, there are certain things that must be done in performing the switch: first, save the address space of the old process as well as the current execution point represented by a portion of the processor state, and secondly, reloading the address space and previous execution point of the new process. The process switch must occur in a specific sequence to insure the new process resumes execution at the same point and in the same logical state as when it was previously switched. In the SYSTEM re-establishing the local memory to its previous state becomes part of the process switch (when switching user processes).

Because of the overhead (unloading and loading all the MMU registers) associated with process switches, provisions are included to make the processes semi-dedicated to a processor and thus make the requirement for memory switches infrequent. In order to make the process switch totally hidden outside the kernel, the segments that were in memory the last time the process was executing must be loaded in memory prior to allowing the process to resume execution. The lack of a "DBR"[6,p.12] is a problem, but saving copies of the MMU, that can be reloaded when required reduces the severity of the problem.

7. Family of Operating Systems

The design in this thesis is not really for a single operating system, but rather for a whole family of operating systems. For any specific system the family member chosen depends on the functions required. A tactical system which is static in nature does not require many of the user services supported by the SYSTEM. For this reason the family member that consists of only the kernel could be the specific operating system chosen for a tactical system. A general purpose time sharing system, on the other hand, is very dynamic in nature, utilizing large address spaces, variable number of users, etc. The family member that supports dynamic linking, a hierarchical file system and demand memory management could be the specific operating system for the general purpose time sharing system.

Operating system sub-setting refers to the ability to form meaningful sub-sets of an operating system. In the design of the SYSTEM a sub-setting capability was one of the goals. The structure is such that many of the services provided by the SYSTEM can be eliminated without effecting the usefulness of the remaining system. That is the SYSTEM can be tailored to fit a number of specific requirements. This is made possible primarily by utilizing a loop free structure[9] within the design. For explanation purposes consider the operating system to be composed of modules. In a loop free structure the dependency is inward or downward (toward the hardware), depending on your point of view. A module only depends on another module at a lower level.

Requiring a loop free dependency structure allows system correctness to be established one module at a time. Modifying a module would only effect the modules above which depend on it.

The design choice to keep the kernel relatively small and put the common user services in the supervisor lends itself to sub-setting. The security kernel would not be changed in any of the sub-sets and thus would not require re-verification. The supervisor supported services (dynamic linking, discretionary security, demand memory management, hierarchical file system) could be removed to meet the needs of the specific use of the system. This makes the sub-sets of the SYSTEM suitable for tactical application, where there is generally no need for demand memory management or dynamic linking (static environment), as well as for general purpose application where all the features can be utilized. It should be noted that any of these meaningful sub-sets would be a secure system since the kernel remains unchanged in every sub-set. Sub-sets of the kernel can also be constructed; however, this would require reverification of the kernel.

8. Levels Of Abstraction

Abstraction is a way of avoiding complexity and a tool by which a finite piece of reasoning can cover a myriad of cases[17]. The purpose of abstracting is not to be vague, but to create a semantic level in which one can be

absolutely precise. Levels of abstraction have been demonstrated to be a powerful design methodology for complex systems. In general, the use of levels of abstraction leads to a better design with greater clarity and fewer errors. A level is defined not only by the abstraction that it supports (for example, a segmented virtual memory) but also by the resources employed to realize that abstraction. Lower levels (closer to the machine) are not aware of the abstractions or resources of higher levels; higher levels may apply the resources of lower levels only by appealing to the functions of the lower levels. This pair of restrictions reduces the number of interactions among parts of a system and makes them more explicit.

Each level of abstraction creates a virtual machine environment. Programs above some level do not need to know how the virtual machine of that level is implemented. For example, if a level of abstraction creates sequential processes and multiplexes one or more hardware processors among them, then at higher levels the number of physical processors in the system is not important. By the rules of abstraction calls to a procedure at a different level must always be made in a downward direction and the corresponding return in the upward direction. Note that at least two of the levels (kernel and supervisor) define virtual machines with rigidly enforced (via hardware) invokation of "extended instruction", i.e. the kernel and supervisor calls.

B. PROPOSED DESIGN

The SYSTEM is composed of two parts, the supervisor and the kernel. The supervisor provides operating system services while the kernel manages physical resources. This division also contributes to the ability to sub-set without affecting the kernel. The supervisor, which consists of procedures, is distributed and exists within the supervisor domain of each user process. The kernel is made up of both procedures and system processes. The procedures are part of the distributed operating system and exist within the kernel domain of each user process. The system processes are not distributed but are separate processes.

1. Notation

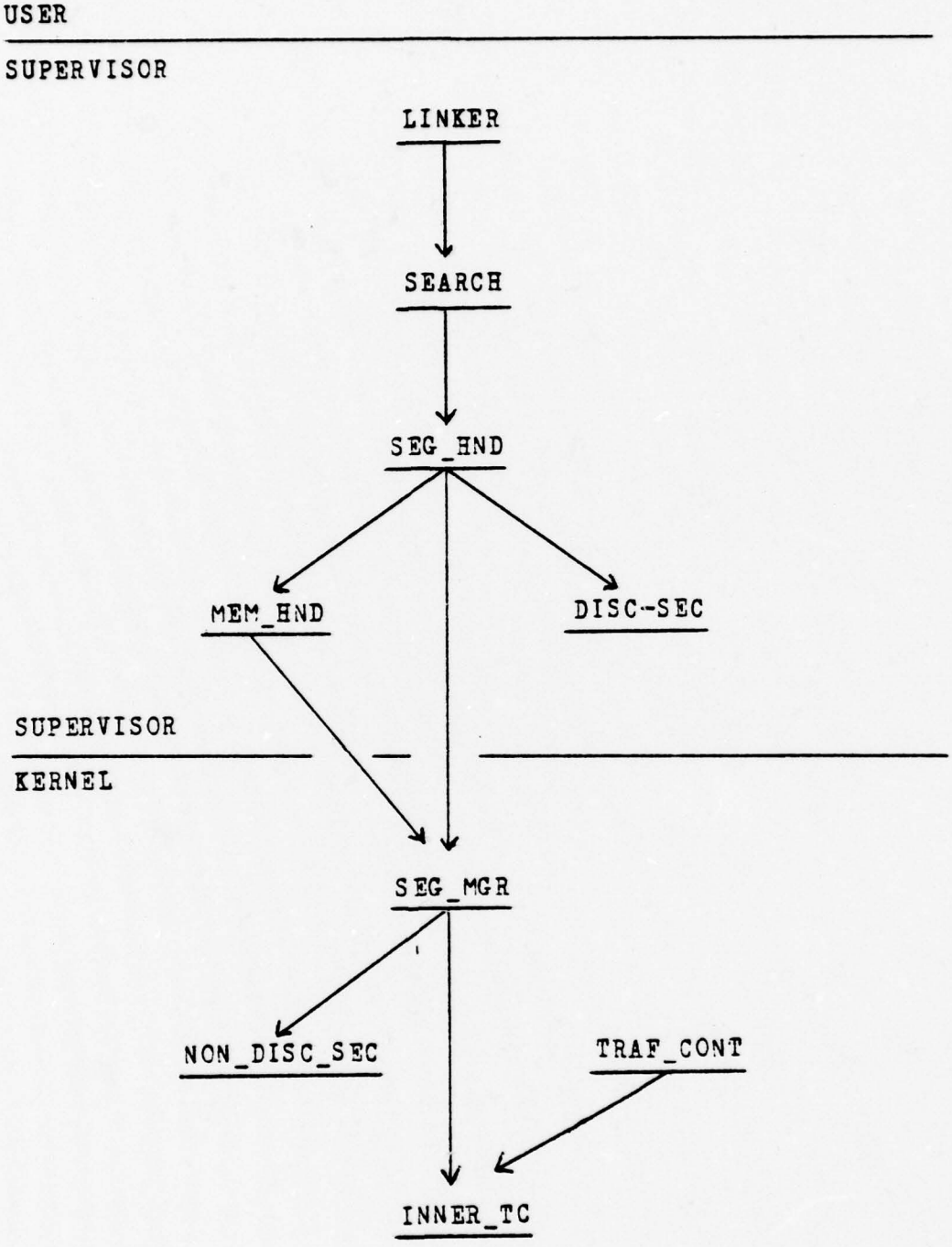
The following is an explanation of the notation used in the following discussions. When a CALL is used the name of the module is given followed by the parameters within parenthesis. When a name in quotes appears as the first parameter in the parantheses it is used to specify the entry within the module. For example CALL INNER_TC('UNLOAD', SEGMENT_#, WRITTEN) the module name is INNER_TC, 'UNLOAD' specifies the entry point and SEGMENT_# and WRITTEN are the parameters. When a SIGNAL is used the first name in quotes specifies the process for whom the signal is intended, the second name in quotes (optional) specifies the specific function requested of that process and the remaining names represent parameters. For example SIGNAL('MEMORY_MANAGER',

'OUT', SEGMENT_#, WRITTEN) the signal is meant for the memory manager process, 'OUT' is the requested function and SEGMENT_# and WRITTEN are parameters. WAIT is used when a process cannot continue execution until it receives a signal from another process. WAIT(PROCESS_ID, MSG). The return parameters PROCESS_ID and MSG are used to indicate the process that sent the signal and the message sent. It should be noted that the above notation is only used to simplify the understanding of what is happening. In an actual implementation the parameters need not be passed in precisely this fashion.

2. System Overview

The following is an overview of the SYSTEM's modules and processes and how they function. Figure 2 represents the modules that exist in the distributed supervisor and the distributed kernel. The levels are used to indicate the dependencies that exist between these modules. The supervisor is made up of four levels of abstraction. It should be noted that all data within the supervisor is per process.

The linker, a level 1 module called LINKER, exists in a segmented virtual memory and provides the mechanisms of dynamic linking. He is invoked by CALL LINKER(SYMBOLIC_NAME). It should be noted that the call could be by link fault as in MULTICS[6]. The linker keeps track of snapped links in the linkage segment (figure 3).



SYSTEM LEVELS

FIGURE 2

The linker utilizes the CALL SEARCH(SYMBOLIC_NAME, SEGMENT_#) to obtain the segment number for unsnapped links.

The searcher, a level 2 module called SEARCH, is invoked by SEARCH(SYMBOLIC_NAME, SEGMENT_#) and is required to return the segment number of the segment specified by the symbolic name. By applying the 'search rules' the symbolic name is converted to a path name in the hierarchical file system. The searcher gets the desired segment number by the CALL SEG_HND(PATH_NAME, SEGMENT_#).

The segment handler, a level 3 module called SEG_HND, is invoked by CALL SEG_HND(PATH_NAME, SEGMENT_#) and is responsible for returning the appropriate segment number. The segment handler utilizes the Segment Table (figure 4) as its data base. To maintain the data base he uses the CALL SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#, SIZE) to the kernel to obtain a segment number for a segment and the CALL DISC_SEC(SEGMENT_#, ENTRY_#, ACCESS) to determine the authorized access (discretionary). The segment handler is also invoked by the virtual faults, SEG_HND('SEG_FAULT', SEGMENT_#) and SEG_HND('MEM_FAULT', SEGMENT_#). The 'SEG_FAULT' is a discretionary security access check and is handled by a CALL DISC_SEC(SEGMENT_#, ENTRY_#). The 'MEM_FAULT' is a request to bring a segment into memory and is handled by a CALL MEM_HND(SEGMENT_#, SIZE).

The memory handler and discretionary security, level 4 modules called MEM_HND and DISC_SEC respectively, are

SYMBOLIC_NAME	SEGMENT_#	OFFSET
TEST1	4	0
TEST2	5	0
THESIS	6	0

LINKAGE SEGMENT (PER SEGMENT)

FIGURE 3

SEGMENT_#	DISC-SEC ACCESS	SIZE	PARENT SEGMENT_#	ENTRY #

SEGMENT TABLE

FIGURE 4

FREE		ALLOCATED		
BASE	SIZE	BASE	SIZE	SEGMENT_#
BASE	SIZE	BASE	SIZE	SEGMENT_#
BASE	SIZE			

MEMORY MAP (LOCAL)

FIGURE 5

invoked by MEM_HND(SEGMENT_#, SIZE) and DISC_SEC(SEGMENT_#, ENTRY_#, ACCESS) respectively. The memory handler provides the dynamic memory management utilizing the Memory Map data base (figure 5). The memory handler uses the CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) in the kernel to bring a segment into memory and the CALL SEG_MGR('SWAP_OUT', SEGMENT_#) to remove a segment. The discretionary security checks the access control lists to determine the authorized access of the process (discretionary).

The distributed kernel is composed of three levels. The segment manager, a kernel level 1 module called SEG_MGR, is invoked by the CALL SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#), CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) and CALL SEG_MGR('SWAP_OUT', SEGMENT_#). The segment manager maintains the Known Segment Table (figure 6) as a per process data base. The segment manager determines allowable access by the CALL NON_DISC_SEC(UNIQUE_ID, ACCESS) and assigns segment numbers by the CALL INNER_TC('ASSIGN', SEGMENT_#, ACCESS). The segment manager brings segments into memory by SIGNAL('MEMORY_MANAGER', 'IN', SEGMENT_#, UNIQUE_ID, BASE_ADDRESS) and removes segments from memory by SIGNAL('MEMORY_MANAGER', 'OUT', SEGMENT_#).

The non-discretionary security, a kernel level 2 module called NON_DISC_SEC, is responsible for determining the authorized access for a given segment. Non-discretionary security is invoked by the CALL NON_DISC_SEC(UNIQUE_ID,

UNIQUE ID	SEGMENT_#

KNOWN SEGMENT TABLE (KST) ENTRY

FIGURE 6

PROCESS_ID	STATE	AFFINITY	PRIORITY	LOC_EX_STATE	VIRTUAL PROCESSOR_#

ACTIVE PROCESS TABLE ENTRY

FIGURE 7

VIRTUAL PROCESSOR_#	VIRTUAL PROCESSOR PRIORITY	PROCESS ID
1	3	
2	7	
3	4	
4	10	

PROCESSOR TABLE

FIGURE 8

ACCESS).

The traffic controller, a kernel level 2 module called TRAFFIC_CONT, is responsible for multiplexing user processes to virtual processors. The traffic controller utilizes the Active Process Table (figure 7) as its data base. traffic controller is invoked by the CALL TRAFFIC_CONT('BLOCK', MSG, WAKING_ID) and CALL TRAFFIC_CONT('WAKEUP', PROCESS_ID, MSG). The traffic controller uses the SIGNAL('MEMORY_MANAGER', 'LOAD', VIRT_MEM_MAP) and SIGNAL ('MEMORY_MANAGER', 'UNLOAD', WRIT_BIT_MAP) to load and unload the processes' segments in memory on the virtual processors. The traffic controller uses the CALL INNER_TC('LOAD_MMU', PROCESS_ID) AND CALL INNER_TC('UNLOAD_MMU') to load or unload the memory management registers of the virtual processors. The traffic controller uses the CALL INNER_TC('IDLE') to remove a virtual processor from contention for rescources. Actually the virtual processor is assigned the lowest priority available and the idle process is loaded.

The inner traffic controller, a kernel level 3 module called INNER_TC, provides the multiplexing of virtual processors to real processors. The inner traffic controller uses the Processor Table (figure 8) as its data base.

The non-distributed kernel consists of two system processes. The memory manager process maintains the Active Segment Table (figure 9) and Global Memory Map (figure 10) as data bases. Basically it loads segments into memory. The

UNIQUE_ID	GLOBAL ADDRESS	WRITTEN BIT	PROCESSOR BIT MAP	CONNECTED PROCESSES	WRITEABLE BIT

AST GLOBAL

VIRTUAL PROCESSOR_ID	SEGMENT_#	UNIQUE_ID	ACCESS	ABS ADDRESS

AST LOCAL (PER PROCESSOR)

ACTIVE SEGMENT TABLE

FIGURE 9

FREE		ALLOCATED		
BASE	SIZE	BASE	SIZE	SEGMENT_#
BASE	SIZE	BASE	SIZE	SEGMENT_#
		BASE	SIZE	SEGMENT_#

GLOBAL MEMORY MAP

FIGURE 10

memory manager process is responsible for putting segments in local/global memory based on user's access.

The I/O manager process processes all the external I/O, this includes I/O to and from the user terminals. The terminals can be thought of as being hard wired. Specific terminals have specific access classes; therefore no kernel passwords are required to determine access class.

The next three sections provide a detailed discussion of the design.

3. Supervisor

The supervisor can be invoked by the following external (user) calls:

SUP_CREATE_SEGMENT(ACCESS_CLASS,SIZE)

SUP_DELETE_SEGMENT(SEGMENT_#)

LINKER(SYMBOLIC_NAME)

SUP_BLOCK(MSG)

SUP_WAKEUP(PROCESS_ID,MSG)

SUP_CREATE_PROCESS(PROCESS_ID, ADDRESS_SPACE)

SUP_DESTROY_PROCESS(PROCESS_ID)

a. Linker (Supervisor)

The linker exists in a segmented virtual memory environment. It is only aware of symbolic names and segment numbers. The choice was made to provide dynamic linking and not assign segment numbers to segments at compile or load time; therefore there is a requirement to resolve external references at run time. In general it is the linker's job to

intervene on a procedure's external references and direct the reference to the appropriate segment. To accomplish this the linker utilizes a "linkage segment" (each process has a linkage segment). The linkage segment contains an entry for each segment known to the process.

Each external reference results in a call to the linker with a parameter that on first reference permits finding the symbolic name of the desired segment.

LINKER(SYMBOLIC_NAME) The linker searches for the entry corresponding to the symbolic name. If found it transfers to the segment number and offset specified in the linkage segment. If not found (first reference) it must first determine the segment number and offset. To obtain the segment number the linker calls the searcher passing as a parameter the symbolic name. SEARCH(SYMBOLIC_NAME, SEGMENT_#) The parameter returned is the segment number. The linker completes the entry in the linkage segment and transfers control to the desired segment.

b. Searcher (Supervisor)

The searcher is aware of the hierarchical file system and a set of search rules. It is involked by SEARCH(SYMBOLIC_NAME, SEGMENT_#). The searcher has the task of resolving a symbolic name into a path name. The searcher recieves as a parameter a symbolic name which is processed and eventually the segment number of the symbolically named segment is returned. To accomplish this the searcher applies the 'search rules'[6]. The search rules are a list of path

names and a simple technique that convert the symbolic name to a path name (note that this is independent of security). The searcher utilizes a calling directory and working directory [6,p.230]. Once the path name is determined the searcher calls the segment handler passing the path name as a parameter. SEG_HND(PATH_NAME, SEGMENT_#) The parameter returned is the segment number. The searcher returns passing the segment number as a parameter to the linker.

c. Segment Handler (Supervisor)

The segment handler understands the hierarchical file system, parent, entry number, access control lists, and segment numbers. The segment handler deals with virtual segment faults (access checks) and virtual memory faults. He is involked by the call SEG_HND(PATH_NAME, SEGMENT_#). The segment handler gets assistance in performing his tasks by utilizing the following calls: MEM_HND(SEGMENT_#, SIZE) to request a segment be put in virtual memory, DISC_SEC(SEGMENT_#, ENTRY_#) a function to determine the authorized access (discretionary security) to a segment, SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#) a kernel call used to determine the segment number and size of the segment indicated by the parent segment number and entry number.

The segment handler maintains a segment table with information that is necessary to control segments at the supervisor level (figure 4). The segment number is unique within the process. Parent segment number is the

segment number of the parent and entry number is the entry within the parent for the segment. Access is that access authorized by the discretionary security policy. Size is the memory required by the segment. The segment handler is required to convert path names to segment numbers as well as to handle virtual segment faults (discretionary security checks) and virtual memory faults. To accomplish these tasks the segment handler has three entry points: SEG_HND, MEM_FAULT and SEG_FAULT.

SEG_HND(PATH_NAME, SEGMENT_#) The segment handler receives as a parameter the path name of the desired segment. One of the design characteristics of the hierarchical file system is that access to a segment requires read access to every segment on the path of the segment. One by one the segments on the path name must be made known and the access established. To do this a recursive algorithm can be utilized that will process each entry within the path name until the path name is resolved. The segment number assigned to the desired segment is returned.

SEG_HND('MEM_FAULT', SEGMENT_#) A virtual memory fault is utilized to support the dynamic memory management outside the kernel. When a segment that is not in memory is referenced a virtual memory fault (hardware initiated, the kernel provides the software interpretation of the fault and provides a transfer vector to the supervisor) is generated to the segment handler. The segment handler uses the Segment

Table to determine the SEGMENT_# and the SIZE of the segment. The memory handler is called, MEM_HND(SEGMENT_#, SIZE).

SEG_HND('SEG_FAULT', SEGMENT_#) A virtual segment fault is used to tell the supervisor that the ACL for the segment referenced has been changed since the last time the segment had been referenced. The segment handler must re-establish the discretionary security. This is done by checking the Segment Table for the parent's segment number and entry number, calling DISC_SEC(SEGMENT_#, ENTRY_#, ACCESS), check the new access, update the Segment Table and return.

d. Memory Handler (Supervisor)

It is the job of the memory handler to provide the dynamic memory management within a fixed size linear virtual memory. The memory handler utilizes two kernel calls 'SWAP_IN' and 'SWAP_OUT' to perform his tasks. SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) is used to request that a segment be brought into memory. SEG_MGR('SWAP_OUT', SEGMENT_#) is used to remove a segment from memory.

The memory handler is tasked by the segment handler to put a segment into memory and provided with the SEGMENT_# and SIZE of this segment. The data base utilized is a Memory Map (figure 5) which indicates free areas and allocated areas. Each process has a memory map which is used to keep track of the virtual memory allocated to the

process.

To provide the demand memory management there are many suitable algorithms[16,p.155]. First fit, best fit and worst fit are among the possible choices for allocating free areas. A least recently used algorithm is generally used for deallocating memory. The used bit is available to provide information to the dealocation scheme. The CALL INNER_TC('GET_USED_BITS', USED_BITS) returns an array of the status of all the used bits. The CALL INNER_TC('SET_USED_BITS', USED_BITS) provides an array of the desired value of the used bits. This provides the mechanism for an approximating efficient Least Recently Used algorithm for dealocation[16]. Allocated areas (figure 5) are identified by (SEGMENT_#, BASE_ADDRESS, SIZE). When tasked, the memory handler searches for a free area large enough for the segment. If there is no free area large enough, the memory handler must utilize the CALL SEG_MGR('SWAP_OUT', SEGMENT_#) to establish a large enough free area. The memory map is updated and the CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) is generated. The memory map is updated and the memory handler returns.

e. Discretionary Security (Supervisor)

This module is only aware of access control lists (figure 11) and how to search one to determine the access to be given the current process. The input parameter is the segment number (of the directory) and entry number of the ACL for the desired segment. The discretionary security

searches the ACL for the PROCESS_ID of the calling process and thereby determines the access, which is returned.

4. Distributed Kernel

There is a gate mechanism (domain change) through which all kernel and supervisor calls pass. Checks are made to determine proper (complete) parameters and the call is directed to the proper module. The kernel is the "priviledged mode" and can execute priviledged instructions. Calls coming from outside the kernel are:

MAKE_KNOWN(PAR-SEG_#, ENTRY_#, ACCESS, SEGMENT_#)

SWAP_IN(SEGMENT_#, BASE_ADDRESS)

SWAP_OUT(SEGMENT_#)

SET_SEG_FAULT(SEGMENT_#)

BLOCK(MSG,WAKING_ID)

WAKEUP(PROCESS_ID, MSG)

CREATE_PROCESS(PROCESS_ID, ADDRESS_SPACE)

START_PROCESS(PROCESS_ID, EXECUTION_POINT)

STOP_PROCESS(PROCESS_ID)

DESTROY_PROCESS(PROCESS_ID)

CREATE_SEGMENT(PAR_SEG_#, ENTRY_#, ACCESS_CLASS, SIZE)

DELETE_SEGMENT(UNIQUE_ID)

INNER_TC('GET_USED_BITS', USED_BITS)

INNER_TC('SET_USED_BITS', USED_BITS)

a. Segment Manager (Kernel)

The segment manager's environment is a segmented

physical memory. The segment manager assigns segment numbers and is responsible for maintaining the status of all segments known to a process. The segment manager's primary data base is the Known Segment Table (KST) (figure 6). The unique_ID is a unique, system wide identifier assigned to each segment. They are assigned from an available list of integers (can be reused when a segment is deleted). Each segment also has an alias that is the unique_ID of and the entry number in its parent. This provides a means of determining the unique_ID of a segment from the segment number of and entry number in the parent.

It should be noted that the reason for the alias is to prevent the unique_ID from leaving the kernel. The alias chosen is derivable from information known to the supervisor, because it relates to the hierarchical file system. This information is per process and not system wide in nature. Although the hierarchical structure of the file system can be derived from the kernel's alias data base, the contention is that the file system in the kernel is a flat one. This method also eliminates the confinement problem. The kernel only requires that the access class of a segment, when created must be at or above the access class of the process creating the segment.

The segment manager can be involked by several calls:

SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#)

SEG_MGR('SET_SEG_FAULT', SEGMENT_#)

SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS)

SEG_MGR('SWAP_OUT', SEGMENT_#)

The CALL SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#). The task is to assign a segment number to the segment specified. PAR_SEG_# and ENTRY_# are the segment number of the parent directory and the entry within that directory. The parent segment number is used to determine the unique_ID of the parent from the KST and this combined with the entry number forms an alias for the desired segment. The segment manager searches the KST to determine if the segment has already been assigned a segment number (already known). If this is the case the segment number already assigned is returned. If the segment is not known then a KST entry must be made. The procedure is as follows: use the PAR_SEG_# and the KST to determine the unique_ID of the parent. Combine the unique_ID of the parent and the entry number to derive the alias of the segment. Use the alias to determine the unique_ID of the desired segment from the alias table (figure 12). CALL NON_DISC_SEC(UNIQUE_ID, ACCESS) to determine the authorized access. The access granted is the desired access or the authorized access, whichever is less. Assign a segment number. Fill in KST entry. CALL INNER_TC('ADD_SEG', SEGMENT_#, ACCESS). Return assigned segment number.

The CALL SEG_MGR('SET_SEG_FAULT', SEGMENT_#). This call is used when the access control list for a segment

FILE	ACL
THESIS	'OCONNELL'(ALL ACCESS), 'RICHARDSON'(ALL ACCESS) 'JONES'(READ ACCESS), 'ALL_OTHERS'(NO ACCESS)

ACCESS CONTROL LIST

FIGURE 11

UNIQUE ID	ALIAS	
	(PARENT UNIQUE_ID)	(ENTRY_#)
15	7	3
13	12	2
7	1	1
12	7	4

ALIAS TABLE

FIGURE 12

MACHINE REGISTERS		
SOFTWARE FAULTS	ACCESS	RELATIVE BASE ADDRESS

LOC_EX_STATE

FIGURE 13

is changed. The segment manager determines the unique_ID of the segment specified and does a SIGNAL('MEMORY_MANAGER', 'SET_SEG_FAULT', UNIQUE_ID).

The CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS). A request to load the specified segment into memory at the indicated base address (relative). The segment manager locates the appropriate KST entry and does a SIGNAL('MEMORY_MANAGER', 'IN', SEGMENT_#, UNIQUE_ID, BASE_ADDRESS) and a WAIT(PROCESS_ID, ABS_ADD, BOUND). The memory manager process loads the segment in memory and returns the absolute address and bound of the segment. The segment manager notifies the inner traffic controller of the update in segment information CALL INNER_TC('LOAD', SEGMENT_#, ABS_ADD, BOUND). The segment manager returns.

The CALL SEG_MGR('SWAP_OUT', SEGMENT_#). The segment manager is tasked with removing the segment from memory. He does a CALL INNER_TC('UNLOAD', SEGMENT_#, WRITTEN) to obtain the value of the written bit and then to unload the segment from memory a SIGNAL('MEMORY_MANAGER', 'OUT', SEGMENT_#, WRITTEN), WAIT('MEMORY_MANAGER') and then returns.

b. Non-Discretionary Security (Kernel)

The purpose of the non-discretionary security is to enforce the non-discretionary security policy by checking the access class of the process against the access class of the desired segment. The access is determined as a result of this comparison. The non-discretionary security module is

invoked by the CALL NON_DISC_SEC(UNIQUE_ID). An algorithm is used for interpreting the lattice for comparing the access classes and determining the authorized access. The non-discretionary security module returns passing the access.

C. Traffic Controller (Kernel)

The job of the traffic controller is to schedule and control processes. The traffic controller utilizes an Active Process Table (system wide) (figure 7) and a Virtual Processor Table (figure 8) to maintain the necessary information about each process. Each virtual processor has a priority (this priority is used by the inner traffic controller when the virtual processors are multiplexed on the physical processors). PROCESS_ID is a unique identifier for each process, which can be mapped to the user. STATE refers to the present state of a process (ready, block, stop, run). AFFINITY is used to specify a binding of a process to a virtual processor either by virtue of dissimilar processor characteristics (strong) or the process has segments in local memory of a processor (weak). PRIORITY is used to determine a scheduling behavior. LOC_EX_STATE provides the means for keeping track of the execution state of the process and is a pointer to a storage area that contains information about the execution state (figure 13).

The traffic controller schedules the processes to run on virtual processors. There is a virtual processor for every loaded process. Each virtual processor has a low

priority process (IDLE) so that the processor is never stopped. The traffic controller provides the BLOCK and WAKEUP functions as a means of providing inter-process communication.

The traffic controller would have a priority driven scheduling algorithm to determine what process to schedule. This could be a simple first come first served algorithm or it could be a complex time sharing algorithm to dynamically change process priority. The method utilized in this thesis is that the traffic controller works on the premise of scheduling the ready process with the highest priority and the proper affinity whenever a virtual processor is available.

Whenever a process blocks itself it is in fact a call to the traffic controller. The traffic controller changes the state of the process to blocked, The traffic controller now has the option of reassigning the virtual processor to another user process or scheduling the idle process (CALL INNER_TC('IDLE')). In the latter case there is no loading or unloading of the process involved and this can be beneficial to control thrashing. Since there are other virtual processors competing for the processor the traffic controller scheduling algorithm will try to leave the process loaded. When the process is put back in the run state it will be in contention for the processor. If another process is to be assigned to the virtual processor then the old process must be unloaded. First the status of the

written bits are determined (CALL INNER_TC('WRITTEN_BITS')). The execution state of the old process is unloaded (CALL INNER_TC('UNLOAD_MMU', PROCESS_ID, LOC_EX_STATE)). SIGNAL('MEMORY_MANAGER', 'UNLOAD', WRIT_BIT_MAP) and WAIT('MEMORY_MANAGER', VIRT_MEM_MAP) are generated, the virtual memory map of the process is returned by the manager process process. The execution state and the virtual memory map of the old process are saved. Now the new process can be loaded. The virtual memory map of the new process is passed to the memory manager process, a SIGNAL('MEMORY_MANAGER', 'LOAD', VIRT_MEM_MAP) and WAIT('MEMORY_MANAGER', ABS_ADD_MAP) are generated. A map indicating the absolute address of the loaded segments is returned by the memory manager process. The execution state of the new process is loaded (CALL INNER_TC('LOAD', LOC_EX_STATE, ABS_ADD_MAP)). This completes the process of switching user processes on a virtual processor.

The TRAFFIC_CONT('WAKEUP'', PROCESS_ID) is also a call to the traffic controller. If the process specified by PROCESS_ID is in the blocked state the traffic controller puts that process in the ready state, he checks the priorities of the running processes and if there is a lower priority process in the run state the virtual processor it is running on is sent a pre-empt interupt CALL INNER_TC('PRE_EMPT_INT', VIRT_PRO_ID) and the traffic controller returns. The pre-empt interupt forces the pre-empted virtual processor to transfer control to the

traffic controller. The traffic controller puts this process in the ready state and then schedules the highest priority process, subject to affinity, as indicated above. If the idle process was running on the virtual processor and if the process loaded in that virtual processor is in the ready state it could be assigned the virtual processor by the CALL INNER_TC('UNIDLE', VIR_PRO_ID). This has the effect of unloading the idle process and loading the process that was previously loaded. It should be noted that except for the special case of the idle process, switching processes is lengthy and, if done too frequently, could lead to thrashing problems.

The traffic controller can be invoked by the calls: 'STOP_PROCESS', 'CREATE_PROCESS', 'START_PROCESS', and 'DESTROY_PROCESS'.

'CREATE_PROCESS', PARAMETER_LIST is used to begin a new process. An entry for the process is made in the active process table.

'STOP_PROCESS' is used to put a process in the STOPPED STATE and the process is removed from the active process table and put in the stopped process table (SPT). The SPT is similar to the APT but it is referenced infrequently.

'START_PROCESS' is used to move a process from the stopped process table (STP) to the active process table and also from the stopped state to the ready state.

'DESTROY_PROCESS' is used to terminate the life

of a process. The process is removed from the APT or SPT and the memory manager process is signaled to disconnect the process from any connected segments.

d. Inner Traffic Controller (Kernel)

The inner traffic controller multiplexes the virtual processors with the physical processors[18]. There is a many to one correspondence from the virtual processors of the traffic controller to the physical processors. In addition there are the virtual processors assigned the system processes. The inner traffic controller uses the data base shown in figure 14. He is also responsible for the mapping registers (hardware segment descriptors) which contain the information shown in figure 15. Each physical processor has only specific virtual processors that can be multiplexed on it. Each virtual processor has a priority and a state (running, ready and wait). The inner traffic controller allows the virtual processor with the highest priority in the ready state to run on the processor. The wait pending bit[3,p.30] is used to avoid a race condition between the signal and wait primitives. The inner traffic controller is able to swap the virtual processors in and out of the processors by loading and unloading the appropriate execution state and mapping registers.

The inner traffic controller provides inner-process as well as intra-process services. He is invoked by a number of calls requesting information contained in the mapping registers or providing information

VIRTUAL PROCESSOR_#	STATE	PRIORITY	"COPY" MMU REG	SOFTWARE FAULTS

PROCESSOR MAP

FIGURE 14

ACCESS BIT	FAULT BIT	WRITTEN BIT	USED BIT	BASE ADDRESS	BOUND

MAPPING REGISTERS

FIGURE 15

to update the mapping registers. To supplement the hardware fault within the memory management registers the inner traffic controller maintains a set of software faults for each segment (segment fault, memory fault). This allows the inner traffic controller to interpret the hardware fault and generate an appropriate virtual fault.

INNER_TC('ASSIGN', SEGMENT_#, ACCESS) - a new segment number has been assigned with the indicated access. Load the appropriate register with the access, set the fault bit and the software memory fault.

INNER_TC('LOAD', SEGMENT_#, ABS_ADD, BOUND) - a segment has been loaded into memory, load the appropriate addresses in the mapping register and reset the memory software fault and fault bit if appropriate.

INNER_TC('UNLOAD', SEGMENT_#, WRITTEN) - the segment is being removed from memory, set the memory software fault and the fault bit and return the value of the written bit.

INNER_TC('WRITTEN_BITS', BITS) - an array reflecting the value of the written bits is returned.

INNER_TC('GET_USED_BITS', USED_BITS) - an array reflecting the value of the used bits is returned.

INNER_TC('SET_USED_BITS', USED_BITS) - an array is received reflecting the desired value of the used bits. The inner traffic controller sets the used bits to the desired values. The hypothesized hardware used bits are also set by hardware whenever a segment is referenced.

INNER_TC('LOAD_MMU', LOC_EX_STATE, ABS_ADD_MAP) - a request to load a virtual processor with a new process and create the memory management unit registers.

INNER_TC('UNLOAD_MMU', LOC_EX_STATE) - a request to unload a virtual processor and save the execution state in the indicated location.

INNER_TC('SET_SEG_FAULT', PROCESS_ID, SEGMENT_#) - a request to set the software segment fault in the data base (figure 14).

INNER_TC('IDLE') - a request to load the idle process and reduce the priority of the virtual processor to the lowest possible.

INNER_TC('PRE_EMPT_INT', VIRT_PRO_ID) - a request to generate a virtual pre_empt interupt to the indicated virtual processor. The inner traffic controller determines which physical processor the virtual processor is in and sends an appropriate hardware interrupt to that processor. If the virtual processor is in the wait state the interupt is held pending until the virtual processor is put in the ready state.

INNER_TC('UNIDLE', VIRT_PRO_ID) - a request to unload the idle process, reinstate the loaded process and restore the priority of the virtual processor.

The inner traffic controller is also invoked by the signal and wait. Signal and wait provide the synchronization between the system processes and the user processes. The inner traffic controller utilizes the signal

and wait primitives to change the state of the virtual processors and thereby control the multiplexing of the virtual processors to the real processors, based on their priorities.

5. Non-Distributed Kernel

The non-distributed kernel consists of the system processes. These processes have the characteristic that they function asynchronous to each user process. The system processes, as they are called, can reside in the local memory of each processor but their shared data bases will reside in global memory.

a. Memory Manager (System Process)

The memory manager process utilizes the Active Segment Table (figure 9) as a data base. The portion of the AST that contains system wide information will reside in global memory. The portion of the AST that only relates to a single processor can be distributed and will reside in local memory.

The memory manager process is responsible for two basic tasks: requests to bring segments into memory and requests to remove segments from memory. Other processes task him by use of the signal and wait primitives. The memory manager process has four tasks (entries): IN, OUT, LOAD, and UNLOAD. The IN and OUT are requests to load and remove a single segment. The LOAD and UNLOAD are requests to load and unload a number of segments.

The task to load a segment requires several considerations. Is the segment currently active (AST entry)? If it is, is it presently residing in global memory? If it is not in global memory does the access of the added process require that it be moved to global memory? How to alert the processes with copies? The AST provides all the necessary answers to render the proper decision as to where to load the segment.

At this time a better look at the AST is called for. It should be noted that every segment that presently resides in memory is active and its address can be determined from the AST. The virtual processor that it is in can also be determined as well as the segment number by which it is known within that virtual processor.

When a segment must be loaded into global memory (based on user access) there is a need to notify processors with a copy, of the segment, of the segments relocation. After the segment has been loaded in global memory, the memory manager process, tasked to load the segment, can determine form the AST in which processors the segment is presently loaded. These processors are sent SIGNAL('MEMORY_MANAGER', 'MOVE', UNIQUE_ID, ABS_ADD) where ABS_ADR is the global address of the segment. Each memory manager process that receives the signal('move') will check his local AST to determine which processes have the segment loaded and the segment number assigned and then CALL INNER_TC('CHANGE_ADD', PROCESS_ID, SEGMENT_#, ABS_ADD) for

each process that has the segment in local memory. The inner traffic controller will update the mapping register to reflect the new absolute address.

If a user requests access, and another user already has write access, there is a need to get the current copy moved to global memory. In this case the memory manager process attempting to load the segment must SIGNAL('MEMORY_MANAGER', 'MOVE_IT', UNIQUE_ID) and WAIT(PROCESS_ID, MSG). The processor with the current copy of the segment was determined from the AST. The memory manager process with the current copy, after receiving the signal('move_it'), will relocate the segment in global memory, CALL INNER_TC('CHANGE_ADD', PROCESS_ID, SEGMENT_#, ABS_ADD) and SIGNAL('MEMORY_MANAGER', 'MOVED', UNIQUE_ID, ABS_ADD). It should be noted that there is some synchronization required between the memory manager process and the inner traffic controller to insure the segment had not been written in during the time it took to move it and change the address.

As segments are loaded and unloaded the AST is updated appropriately. When a segment is removed from memory if it has been written in the segment is copied back to secondary storage.

The AST also provides a method of notifying processes of segment faults. If the memory manager process (for each processor connected with a loaded connected process) is notified when the access control list for a

segment is changed by SIGNAL('MEMORY_MANAGER', 'SET_SEG_FAULT', UNIQUE_ID) then every loaded connected process can be notified by CALL INNER_TC('SET_SEG_FAULT', PROCESS_ID, SEGMENT_#). For processes that are not loaded, the traffic controller is similarly called to set the software segment fault (figure 13). This means that the software segment fault will have to be set for connected processes when a segment is removed from the AST.

b. I/O Manager

The I/O manager is responsible for the external I/O. There could be more than one I/O manager process, conceivably one for each external device; corresponding kernel calls must be provided. For example there could be an I/O manager that handles all the external I/O to and from the user terminals. It is sufficent, at this point, to say that the I/O manager exists and handles external I/O.

6. Follow On Work

It should be re-emphasized that this is a design and not an implementation. Although the detail is left for further work, the design proposed forms a substantial basis upon which an implementation can be realized. The system process structure is provided for in the design; however, the system processes have been treated lightly and require additional work. The user interface (supervisor calls) presented is by no means an exhaustive list and could use further extension for additional supervisor services.

IV. CONCLUSION

The state of the art techniques and design methodology used to design secure operating system for multiple mini and maxi processors have been found applicable to the multiple microprocessor environment. The principal conclusion is that the operating system design in this thesis will make it possible to more effectively use modern microprocessors than has been possible in the past.

One question that is addressed concerns the operating system's ability to scale. Systems now available can support four or five microprocessors. Increasing that number of microprocessors quickly brings serious degradation because of the increased bus contention. The expected scaling factor is much better for this design. The bus contention has been significantly reduced - segmentation permits effectively using local memory instead of global memory.

This design supports a family of operating systems, not just one designed for a specific application. Sub-sets of this system can be constructed to provide the desired functions because the design used a loop free structure. Included family members range from a core resident tactical system to a virtual memory time sharing system.

Configuration independence is supported in this design. One or many physical processors can be added or subtracted from the system without affecting the workability of the

system. Similarly memory can be added or subtracted.

Security has been designed into this system. It was not added on as an afterthought. This design used a security kernel based upon a mathematical model to insure the security. A secure multilevel environment is provided by this system.

Commercial devices will soon be widely available to implement this operating system. The Zilog Z8000 series, microprocessor, for example will provide the segmentation and multiple domains necessary for an effective system. The present data buses are compatible and when used with this operating system allow a significant number of processors to be effectively used.

LIST OF REFERENCES

1. "Architecture of a New Microprocessor", Computer, v.12 No 2, p.10, February 1979.

2. Mitre Corporation Report 2934, The Design and Specification Of A Security Kernel for the PDP-11/45, by W.L. Schiller, May 1975.

3. Saltzer, J.H., Traffic Control in a Multiplexed Computer System, Ph.D.Thesis, Massachusetts Institute of Technology, 1966.

4. Lipner, S.B., "A Comment On The Confinement Property", Operating System Review, v.9, p.192-195, November 1975.

5. Schroeder, M.D., "A Hardware Architecture for Implementing Protection Rings", Communications of the ACM, v.15 No. 3, p.157-170, March 1972.

6. Organick, E.I., The Multics System: An Examination of Its Structure, MIT Press, 1972.

7. Dijkstra, E.W., "The Structure of the 'THE' Multiprogramming System", Communications of the ACM, v.11, p.341-346, May 1968.

8. Janson, P.A., "Dynamic Linking And Environment Initialization In A Multi-Domain Process", Operating System Review, v.9 No. 5, p.43-50, November 1975.

9. Schroeder, M.D., Clark, D.D., and Saltzer, J.H., The Multics Kernel Design Project, paper presented at ACM Symposium, November 1977.

10. LtCol Schell, R.R., "Computer Security: The Achilles' Heel of the Electronic Air Force?", Air University Review, v.XXX No.2, January 1979.

11. Millen, J.K., "Security Kernel Validation In Practice", Communications of the ACM, v.19, p.244-250, May 1976.

12. Denning, D.E., "A Lattice Model Of Secure Information Flow", Communications of the ACM, v.19, p.236-242, May 1976.

13. Mitre Corporation Report ESD-TR-73-278, v.2, Secure Computer Systems: A Mathematical Model, by L.J. Lapadula and D.E. Bell, November 1973.

14. Mitre Corporation MTR-2932, A Software Validation Technique for Certification, Part I: The Methodology, by Bell, D.E. and Burke, E.L., November 1974.

15. Honeywell, Multics Processor Manual, p.8-1, Order Number AL39, Rev. 0, April 1976.

16. Madnick, S.E. and Donovan, J.J., Operating Systems, McGraw Hill, 1974.

17. Dijkstra, E.W., 'The Humble Programmer', Communications of the ACM, v.15, p.859-866, October 1972.

18. Reed, D.P., Processor Multiplexing In A Layered Operating System, Master's Thesis, Massachusetts Institute of Technology, MIT/LCS/TR-164, 1976.

19. Janson, P.A., Using Type Extension To Organize Virtual Memory Mechanisms, Ph.D. Thesis, Massachusetts Institute of Technology, MIT/LCS/TR-167, 1976.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Lt.Col. R. R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	10
5.	Lt. L. D. Richardson, USN 1312 Pearl Street Onawa, Iowa 51040	2
6.	Capt. J. S. O'Connell, USMC 1817 Chestnut Street Alameda, California 94501	2
7.	LT(JG) Luis A. Guillen Salazar #120, Barranco Lima, Peru South America	1